Encapsulating Nondeterminacy in an Abstract Data Type with Determinant Semantics [1]

F. Warren Burton
School of Computing Science,
Simon Fraser University
Burnaby, British Columbia,
Canada V5A 1S6

burton@cs.sfu.ca

January 2, 2007

[1]This work was supported by the Natural Science and Engineering Research Council of Canada. An earlier version of this paper appeared in the *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*.

Abstract

A parallel program may be indeterminate so that it can adapt its behavior to the number of processors available.

Indeterminate programs are hard to write, understand, modify or verify. They are impossible to debug, since they may not behave the same from one run to the next.

We propose a new construct, a polymorphic abstract data type called an *improving value*, with operations that have indeterminate behavior but simple determinate semantics. These operations allow the type of indeterminate behavior required by many parallel algorithms.

We define improving values in the context of a functional programming language, but the technique can be used in procedural programs as well.

1 INTRODUCTION

A parallel program may be indeterminate so that it can adapt its behavior to the number of processors available. For example, in a combinatorial search, many different processes may be searching different subspaces in parallel. These processes may all access and update a global variable that gives information on the best solution found so far. The current value of the variable may be used to determine if a subspace may be pruned from the search. Since the processes are not synchronized, the pruning of a particular subspace may depend on when a shared variable is read. This causes indeterminate behavior. The overall result of the program may or may not be determinate.

Indeterminate programs are hard to write, understand, modify or verify. They are impossible to debug, since they may not behave the same from one run to the next.

We propose a new construct, a polymorphic abstract data type called an *improving value*, with operations that have indeterminate behavior but simple determinate semantics. These operations allow the type of indeterminate behavior required by many parallel algorithms. Operationally, we may know a lower bound (or upper bound) for an improving value at any given time. If this bound is sufficient, we may act on it. Otherwise, the bound may improve as the computation proceeds.

We define improving values in the context of a functional programming language, but the technique can be used in procedural programs as well. We will use the notation of the Miranda[1] functional programming language [10, ?, 11].

In section 2 we will briefly review the concept of speculative evaluation. The *improving value* abstract data type will be introduced in section 3. Several examples of the use of improving values are given in section 4, including a parallel least-cost search algorithm that is only seven line long. In these three

[1]Miranda is a trademark of Research Software Ltd.

sections we will limit our attention to two types of computing devices: Those with a single processor and those with an infinite number of processors. Since machines of the second type are not yet on the market, in section 5 we will consider how to make best use of a limited number of processors. The least-cost search algorithm will again be considered. An axiomatic definition of improving values is given in section 6, along with several properties of improving values, and an outline for a simple correctness proof for the least-cost search algorithm. Section 7 is the conclusion.

2 SPECULATIVE EVALUATION

Often it is necessary for a good parallel algorithm to perform more work in total than would be performed by a good sequential algorithm for the same problem. Sometimes the best sequential algorithm may not have a very high potential for parallelism, so a different algorithm must be used. In other cases, the sequential algorithm may have a high potential for parallelism, but only if some work is done before it is known to be required. We are interested in algorithms of the second kind.

Recall that, by definition, any problem in NP can be solved by a nondeterministic Turing machine in polynomial time. Consider any NP-complete problem, and any nondeterministic polynomial time algorithm to solve the problem where none of the nondeterministic choices lead to a nonterminating computation. A sequential algorithm for this problem can be produced by simulating the nondeterminism by backtracking. In the worst case, we will have to do an exhaustive search using the backtracking algorithm. No other sequential algorithm can solve this problem in polynomial time in the worst case, assuming P $\neq$ NP.

With unbounded parallelism, we can solve any NP-complete problem in polynomial time by considering all of the alternatives of the nondeterministic choices

in parallel. Since no sequential algorithm for any NP-complete problem has a worst case polynomial time solution (assuming P $\neq$ NP), all NP-complete problems must have a high potential for parallelism. On the other hand, a sequential backtracking algorithm could go directly to the solution, solving the problem as quickly as with unbounded parallelism, in the best case. Hence we have an example of a problem where we can gain speed through the use of parallelism, in the average case, but only if we are willing to perform some work that may not be required.

The use of *speculative evaluation* [1, 2, 3] has been proposed for problems such as this. A speculative computation is a computation that may or may not be required later. For example, while considering one alternative in a backtracking algorithm, speculative computations may be exploring other alternatives.

It is important that a programmer be able to assign priorities to speculative computations, since some but not all speculative computation normally will be needed eventually. For example, in a speculative backtracking algorithm, one speculative computation should have a higher priority than another if it is searching further to the left in the backtracking tree.

All computation that is not speculative is called *mandatory*. If a speculative computation is found not to be required, then it may be terminated. If a speculative computation is found to be required, then it must be upgraded to mandatory. Mandatory computation must be favored over speculative computation, at least to the extent that some mandatory computation is always progressing. Higher priority speculative computation should be favored over lower priority speculative computation. In the degenerate case of a single processor, no speculative computation will ever be performed. With unbounded parallelism, all speculative computation will be performed, at least until it is found to be not required.

We note that the problem of locating and terminating unneeded computation may be expensive, so speculative computation is probably advisable only for

highly parallel systems. A technique similar to garbage collection may be used to collect unneeded processes [6, 8].

Another approach to unneeded speculative computation is to let speculative computation continue to run until the program as a whole is ready to terminate. At this time all remaining speculative computation can be terminated at once. If no unneeded speculative computation has a priority as high as any needed speculative computation, and no speculative computation is ever run when either a mandatory computation or higher priority speculative computation could be run instead, then aside from overheads and storage costs, there is no harm in letting speculative computation continue to run. However, the overheads and storage costs may be sufficient to argue for the former method. In any case, unneeded speculative computation does not need to be terminate with any particular speed provided speculative computations have been assigned appropriate priority values.

A single function *spec* is sufficient to introduce speculative evaluation. The type of spec is

$$spec :: (* \text{ -> } **) \text{ -> } (* \text{ -> } **)$$

Semantically *spec* is the identity function restricted to functions. That is

$$spec\ f\ x = f\ x$$

Operationally, *spec* will initiate the speculative evaluation of its second argument before applying its first argument to its second.

Two useful functions are defined in Fig. 1. The function *spec_or* is defined in terms of the *conditional or* operator, $\vee$. When applied, *spec_or* will evaluate its first argument as a mandatory computation and its second argument as a speculative computation. It the first argument returns *True* then the speculative computation may be terminated by the implementation. If the first argument evaluates to *False* then the speculative evaluation must be upgraded to mandatory if it has not already terminated. This function could be useful in a simple

```
spec_or a b = spec (or a) b
              where
              or a b = a ∨ b


start [ ] = [ ]
start (x:xs) = spec (spec cons x) (start xs)
               where
               cons a b = a:b
```

Figure 1: Two useful functions for initiating speculative computation.

backtracking algorithm that returns a boolean result. The function *start* will initiate the speculative evaluation of all of the elements in the list to which it is applied. In Miranda, the colon is an infix cons operator.

Examples of speculative algorithms may be found in [2].

3 IMPROVING VALUES

Often in combinatorial search algorithms, bounds are maintained to help decide when pruning is possible. For example, in a branch-and-bound search for a least cost solution, a program may keep the value of the best solution found so far. If it is possible to establish that a subspace cannot return a better solution, then the subspace can be pruned. Alpha-beta search, and similar algorithms, also maintain bounds. In all of these cases, the values of the bounds change monotonically as the computation progresses. We will limit our attention to lower bounds. Upper bounds can be handled in a similar manner.

We proposed the use of the polymorphic abstract data type *improving* for a

```
abstype improving *
with
make      :: * -> improving *
break     :: improving * -> *
minimum   :: improving * -> improving * -> improving *
spec_max  :: improving * -> improving * -> improving *
```

Figure 2: Signature of type *improving*.

lower bound that may improve (become a tighter bound) over time. The type signature is given in Fig. 2.

The functions *make* and *break* are type transfer functions, and *minimum* and *spec_max* compute the minimum and maximum of *improving* values, respectively, with some added laziness.

The function *minimum* is strict in both its arguments, but *spec_max* is strict only in its first argument. The second argument of *spec_max* is evaluated as a speculative computation. We will be able to use the result returned by *spec_max* before its second argument has been evaluated if its first argument provides sufficient information, just as we are able to use the result returned by *spec_or* before its second argument has been evaluated provided its first argument is *True*. In general, once the first argument of *spec_max* has been evaluated, we will have a lower bound on the result. If this is not sufficient, we can wait for the bound to improve, which will happen when the value of the second argument is available.

For example,

$$break\ (minimum\ (make\ 5)\ (spec\_max\ (make\ 7)\ \bot)) = 5$$

Here $\bot$ denotes an undefined value or a nonterminating computation. We cannot test a value for equality with $\bot$, since the halting problem is undecidable.

In practice, we are not really concerned with handling nonterminating computations or undefined values. However, if any subexpression of an expression can be replaced by $\perp$ without changing the value of the expression, then the subexpression need not be evaluated. We will refer to the value of (*spec_max* (*make* 7) $\perp$) as "at least 7". In general, (*spec_max* (*make a*) *b*) may be approximated by "at least *a*" = (*spec_max* (*make a*) $\perp$). If this approximation is sufficient for our purposes, then *b* need never be evaluated. If we need a better approximation, then *b* will need to be evaluated. By analogy, when computing (*hd* (*a*:*b*)), the value of *b* is not required because an approximation to a list, with the first element defined, is all that the head function, *hd*, requires.

The definition of *improving* values is asymmetric in two ways. First, *spec_max* is asymmetric in its arguments. That is, it is not communitive when viewed as a binary operation For example,

$$break\ (minimum\ (make\ 5)\ (spec\_max\ \perp\ (make\ 7))) = \perp$$

It would be nice if this expression would evaluate to 5. However, to implement this would require the implementation of nonsequential functions. An example of a nonsequential function is the *parallel_or* function, defined so that (*parallel_or True* $\perp$) = (*parallel_or* $\perp$ *True*) = *True*. Since *parallel_or a b* must return *True* if either *a* or *b* evaluate to *True*, both arguments must be evaluated in parallel with fair scheduling, or at least concurrently. This tends to be much more expensive than speculative evaluation.

If we had a *parallel_max* operation, defined so that

$$break\ (minimum\ (make\ 5)\ (parallel\_max\ (make\ 7)\ \perp)) = 5$$

and

$$break\ (minimum\ (make\ 5)\ (parallel\_max\ \perp\ (make\ 7))) = 5$$

then we could define *parallel_or* by

parallel_or a b

$= break\ (minimum\ (make\ True)\ (parallel\_max\ (make\ a)\ (make\ b)))$

assuming that *False* $<$ *True*. Notice that as soon as either a or b is know to be *True*, we have sufficient information to arrive at the overall result. The use of *minimum* is required since a polymorphic *parallel_max* operation would not know that *True* is the largest possible boolean value.

The second way in which improving values are asymmetric is that *minimum* is strict in both of its arguments, while *spec_max* is strict only in its first argument. For example,

$$break\ (spec\_max\ (make\ 5)\ (minimum\ (make\ 2)\ \bot)) = \bot$$

Again, it would be nice if we could restore symmetry and produce a more general abstract data type. Suppose we replace *minimum* with a *spec_min* operations, and that *spec_min a b* will return an improved improving value corresponding to the minimum of a and b, but with the further property that we need not evaluate b if a gives sufficient information. The problem with this is that we can construct a general minimax tree. The best parallel evaluation strategy for minimax trees is still an active research area. The most common sequential method for evaluating minimax trees is to use a sequential alpha-beta search algorithm. When the above expression

$$break\ (spec\_min\ (spec\_max\ a\ b)\ (spec\_max\ c\ d))$$

is evaluated using an alpha-beta search, first *a, b* and *c* will be evaluated, and then, if required, *d* will be evaluated. However, with our asymmetric operators (i.e. *minimum* rather than *spec_min*),

$$break\ (minimum\ (spec\_max\ a\ b)\ (spec\_max\ c\ d))$$

both a and c will be evaluated initially, since *minimum* is strict in both of its arguments and *spec_max* is strict in its first argument. If the value of a is (*make*

5) and the value of c is (*make* 2), then clearly the value of d will be required. If d is (*make* 3), then the value of the overall expression can be computed as 3, without evaluating b. In this case our evaluation strategy is more efficient than alpha-beta. We do not claim that our operators are better in general for searching minimax trees, but argue that using alpha-beta search to implement a more general *improving* abstract data type is not what we want. In general, our approach is simpler than alpha-beta, and powerful enough to implement alpha-beta (see section 4).

As a general rule, the right argument of an application of *spec_max* need not be evaluated if the overall result of the computation containing the application can be computed without this value. There are some minor exceptions. For example, the value of

break (*minimum* (*spec_max* (*make* 5) $\perp$) (*spec_max* (*make* 5) (*make* 4)))

will be $\perp$. The reason for this is that *minimum* is communitive. In order to look at the 4 without being caught by the $\perp$, *minimum* would have to be either nonsequential or noncommunitive. Similarly,

break (*minimum* (*spec_max* (*make* 5) $\perp$) (*make* 5))

is $\perp$, since we want

spec_max (*make* 5) (*make* 4) = *make* 5

to hold. A precise formal definition of the *improving* value operators is given in section 6.

In a typical application of improving values, *spec_max* is likely to be applied to a pair of arguments where the left argument is known to be a lower bound for the right argument. The left argument will often be of the form (*make* a), while the right argument will usually be a complicated recursive expression. If the lower bound given by the left argument is large enough, then the right argument will not need to be evaluated. This is how pruning occurs in combinatorial searches.

An improving value can be represented by a strictly increasing list of lower bounds. The final element of the list will be the true value that previous list elements bound. Infinite lists and partial lists will be considered in section 6. We will assume that for any two values a and b, the value of $a < b$ is defined. This assumption is discussed further in section 6.

With this representation, the *improving* abstract data type can be implemented as shown in Fig. 3. The function *make* produces a singleton list, and *break* returns the last element of a list. The *spec_max* function starts the speculative evaluation of its second argument and then appends its arguments, removing any values in the second list that are less than or equal to the final element of the first list. This ensures that the list of approximations remain strictly increasing. On the other hand, *minimum* merges two lists, removing duplicates in order to maintain strict monotonicity. In addition, the merge ends as soon as either list ends. Notice how the second and third equations defining *short_merge* discard the value of one of the arguments. This is where the pruning of unneeded computation takes place. For example,

$$\mathit{minimum}\ (\mathit{make}\ 4)\ (\mathit{spec\_max}\ (\mathit{make}\ 3)\ (\mathit{make}\ 5))$$

will cause the lists [4] and [3, 5] to be merged to produce [3, 4], with the 5 thrown away. As a second example,

$$\mathit{minimum}\ (\mathit{make}\ 4)\ (\mathit{spec\_max}\ (\mathit{make}\ 5)\ \bot)$$

will be represented by the singleton list [4]. The 5 will cause the second list to be discarded before the $\bot$ is encountered.

We will sometimes refer to the elements of a list representing an *improving* value as a *progress report*. Each progress report gives some new information about the final value of an *improving* value. An *improving* value will continue to produce progress reports until either some progress report contains sufficient information or a final value for the *improving* value is produced. It is easy to see that every element of a list representing an *improving* value, except for the

$$improving\ *\ ==\ [*]$$

$$make\ a = [a]$$

$$break\ x = last\ x$$

$$spec\_max\ xs\ ys = spec\ (monotonic\_append\ xs)\ ys$$

$$minimum\ xs\ ys = short\_merge\ xs\ ys$$

$$monotonic\_append\ xs\ ys = xs\ ++\ dropwhile\ (\leq last\ xs)\ ys$$

$$\begin{aligned}
&short_merge\ [\,]\ [\,] = [\,]\\
&short_merge\ (x{:}xs)\ [\,] = [\,]\\
&short_merge\ [\,]\ (y{:}ys) = [\,]\\
&short_merge\ (x{:}xs)\ (y{:}ys)\\
&\quad = x{:}short_merge\ xs\ ys, \qquad \textbf{if}\ x = y\\
&\quad = x{:}short_merge\ xs\ (y{:}ys), \textbf{if}\ x < y\\
&\quad = y{:}short_merge\ (x{:}xs)\ ys, \textbf{if}\ x > y
\end{aligned}$$

Figure 3: A simple implementation of *improving*.

final element of the list, must at some time have occurred in a left argument of an application of *spec_max*. This property is trivially true of the singleton lists produced by *make*, and is preserved by *minimum* and *spec_max*. Hence we may also refer to a value in the left argument of a call to *spec_max* as a progress report.

If the abstract type *improving* is implemented as a language primitive, then a more efficient implementation than the one described above is possible. Each *improving* value is represented by a pair consisting of the best approximation found so far and a flag indicating whether this is a final value. Each *improving* value has an associated process that will update the approximation as required and supply the latest value to other processes upon request. This save storing more than one value and saves other processes the cost of examining out of date values before coming to the most recent value. In some cases this can significantly reduce the amount of communication between processes. However, if this approach is to retain deterministic semantics, then it is necessary that the domain of values used in improving value be a flat domain, as discussed in section 6, and that the order used be a total order, for example so that distinct nodes with identical cost be ordered in a unique way, regardless of which is encountered first. This can be enforced by making *make* force total evaluation of its argument, as outlined in section 6, and allowing only the standard "<" relation to be used in *improving* values.

A dual abstract data type, *improving'* may be defined using a ">" comparison rather than "<". This abstract data type will have operations *make'*, *break'*, *maximum'* and *spec_min'*. In [4] it was suggested that the *improving* abstract type be parameterized with an order relation. However, if other than a total order is used, then the theoretical foundations discussed later fail to hold, and the improved implementation suggested above may produce nondeterministic results.

If we want a type *improving* t, for some type t, except we want to use an

order relation **R** rather than "<", then it is sufficient to define a function $f :: t$ -> t', for some type t', such that $(f\ x) < (f\ y)$ holds whenever x **R** y does. It is now possible to pair each value x with $f\ x$ and use the type *improving* (t', t), since $(f\ x, x) < (f\ x, y)$ whenever $(f\ x) < (f\ y)$ with the standard "<" comparison in Miranda. On the other hand, if $(f\ x) = (f\ y)$ then $(f\ x, x) < (f\ x, y)$ will hold if and only if $x < y$ with the default ordering on type t. Two improving values are "equal" only if they are identical.

One other obvious extension to the *improving* value abstract data type would be the addition of comparison operators, such as *less* defined by

less [ ] [ ] = False
less (x:xs) [ ] = False
less [ ] (y:ys) = True
less (x:xs) (y:ys)
$= x$:*less* xs ys, **if** $x = y$
$= x$:*less* xs (y:ys), **if** $x < y$
$= y$:*less* (x:xs) ys, **if** $x > y$

If we wanted a "less than or equal to" test, we could replace the first line with

less_equal [ ] [ ] = True

and change names throughout the rest of the function definition. An early version of *improving* values included comparison operations. However, we found we never used them.

4 EXAMPLES

In this section we will consider parallel versions of three search algorithms.

4.1 Least-Cost Search

With many combinatorial problems it is necessary to search a solution space for the best solution to the problem. We will assume that "best" means smallest and call the measure of a solution the *cost* of the solution. Horowitz and Sahni present a least-cost search algorithm for this problem [7]. Assuming the costs of leaves and lower bounds for internal nodes are all distinct, this algorithms expands exactly the same nodes as the A^* algorithm in the case of a single processor, which is optimal [9].

We will assume that the solution space is organized as a tree with all possible solutions at leaves. For any node, *node*, *is_leaf node* is a *bool* indicating whether or not the node is a leaf. For any leaf node, *cost node* is the cost of the solution. If the node is not a leaf, then *children node* is a list of the children of *node* and *lower_bound node* is a lower bound on the least cost solution to be found in the subtree rooted at *node*. We will assume that any node that is not a leaf will have at least one child. Finally, *nil_node* is a special node such that *nil_node* $<$ *node* for any node, *node*, that may be encountered during a search. Otherwise nodes are ordered arbitrarily.

Let us first consider a simple exhaustive search algorithm to solve this problem. Such an algorithm is given in Fig. 4. The function returns an ordered pair, consisting of the cost of the best solution together with that solution. Note that if *cost1* $<$ *cost2* then (*cost1, node1*) $<$ (*cost2, node2*) in Miranda. We should note that "." is an infix function composition operation and *foldr1* is a function that will apply a binary function to elements of a list, reducing the list to a single value (e.g. *foldr1* (+) *xs* will compute the sum of the elements of the list *xs*). We will use *mini* and *maxi* for binary minimum and maximum functions, respectively. The function *map* will apply a function individually to elements of a list, producing a new list.

Fig. 5 gives an equivalent (but slightly less efficient) version of this search. We know that *lower_bound root* is a lower bound on the cost of the node returned

```
search root
    = (cost root, root), if is_leaf root
    = subtree_search,    otherwise
      where
      subtree_search = (foldr1 mini.map search.children)root
```

Figure 4: An exhaustive search algorithm.

```
search root
    = (cost root, root),              if is_leaf root
    = maxi bound subtree_search, otherwise
      where
      bound = (lower_bound root, nil_node)
      subtree_search = (foldr1 mini.map search.children) root
```

Figure 5: A modified exhaustive search algorithm.

by searching the subtree rooted at *root* and, in case it is a tight lower bound, *nil_node* is less than any other node. It follows that

(*lower_bound root, nil_node*) < (*foldr1 mini.map search.children*) *root*

so the two algorithms must return the same result.

A minor further modification, to introduce *improving* values, yields the least-cost search algorithm in Fig. 6. Notice how the search of each subtree starts by providing a progress report in the form of a lower bound on the best solution to be found in the subtree. If a better solution has been found anywhere in the tree, this progress report is sufficient for the subtree to be pruned.

In the case of a single processor, where no speculative evaluation is performed until it becomes mandatory, only those nodes with a lower bound less than or

```
search = break.search'

search' root
    = make (cost root, root),          if is_leaf root
    = spec_max bound subtree_search, otherwise
      where
      bound = (make (lower_bound root, nil_node))
      subtree_search = (foldr1 minimum.map search'.children) root
```

Figure 6: A least-cost search algorithm.

equal to the cost of the optimal solution can ever be expanded. Hence, in the sequential case this algorithm is optimal with respect to the number of nodes expanded, assuming all costs and lower bounds are distinct. With an unbounded number of processors, all paths are searched in parallel, at least until a least cost solution is found.

We should note that we have associated an improving value with each internal node in the tree we are searching. Each pruned node is pruned at the level where a better solution exists in the subtree rooted at its sibling node. If each *improving* value is represented by the best approximation found so far, and a flag indicating whether this is a final value, as suggested at the end of section 3, then this approach can considerable reduce communication bottlenecks.

On the other hand, some branch-and-bound algorithms may maintain a single global bound, which can be a bottleneck in a massively parallel computer systems. It does not appear to be possible to use a single global bound with the *improving* value approach. The reason is that the *improving* value representing the global optimal value depends on the search space as a whole. The final value cannot be know until the search has been completed. On the other hand, if a node is not to be pruned, we need the final value of the global bound to

establish this fact, because if the global bound is still improving, it can improve to the point where the subtree in question can be pruned. In other words, we need the overall result to determine if a local search can be pruned, but need the local search result to compute the overall result if the local search is not to be pruned.

4.2 Breadth-First Search

Breadth-first search can be considered a special case of least-cost search, where the cost of a solution is its depth in the tree. We will consider a breadth-first search where a subtree may contain no solutions and solutions may be found at other than leaf nodes. We will require that the left-most solution node be returned in case there are several solutions at the same depth.

Fig. 7 shows a parallel breadth-first search algorithm. The algorithm takes a node as a parameter and returns a pair consisting of the depth of a least deep solution and the solution itself. We assume that we are given two functions: *is_solution*, which determines whether a node is a solution to the problem of interest, and *children*, which will generate a list of the children of a node. As with the least-cost search algorithm, we assume the existence of a node *nil_node*. We also assume the existence of a value *infinity* which is greater than the depth of any reasonable search. This simplifies the algorithm and allows us to use the built in order relation "<". Notice that we have used the function *start* defined in Fig. 1 to initiate the parallel searching of all subtrees when the subtree root has been found not to be a solution. A subtree is pruned whenever a progress report is sufficient to eliminate it from consideration. That is, if the local search is at a greater depth than a know solution, it is terminated.

4.3 Alpha-Beta Search

As a final example we will consider a parallel alpha-beta search algorithm for searching a game tree, based on the sequential alpha-beta search algorithm

```
breadth_first = break.search 0
search depth root
    = make (depth, root),                    if is_solution root
    = make (infinity, nil_node),             if kids = [ ]
    = spec_max progress_report solution,     otherwise
      where
      kids = children root
      progress_report = make (depth, nil_node)
      solution = foldr1 minimum (start [search (depth + 1) k | k <- kids])
```

Figure 7: A parallel breadth-first search algorithm.

given in [7]. For simplicity, the algorithm returns the value of the best move, not the move itself. Again *start* is used to initiate the searching of subtrees. The algorithm is given in Fig. 8.

We assume the existence of three functions. The function *is_leaf* determines whether a node is a leaf. For leaf nodes *eval* computes the value of the node to the player whose turn it is, and for other nodes *children* computes the children of the node (of which we assume that there is at least one.)

The function *scan* is a commonly used function, similar to *foldr1*, but returning a list of "partial sums" rather than just the final "sum". It is defined by

```
scan f a xs
    = a : scan' f a xs
      where
      scan' f a [ ] = [ ]
      scan' f a (x:xs) = scan f (f a x) xs
```

The function *zip2* maps two lists into a list of corresponding pairs, ending as

```
alpha_beta root alpha beta
  = eval root,                                  if is_leaf root
  = break (minimum (make beta) best), otherwise
    where
    best = (foldr1 spec_max.map make) alphas
    alphas = spec (scan maxi alpha) (start searches)
    searches = [ - (alpha_beta child ( - beta) ( - new_alpha)) |
                   (child, new_alpha) <- zip2 (children root) alphas]
```

Figure 8: An alpha-beta search algorithm.

soon as either list ends.

The expression (*alpha_beta node alpha beta*) searches for the value of the best move for the player whose turn it is, subject to the constraint that only moves with value between *alpha* and *beta* are considered. We know that by making a different move, we can get to a position with value at least *alpha*, and also know that our opponent can keep us from getting to a position of value greater than *beta* by making a different move earlier. The initial call is of the form

alpha_beta root $(-\ m)\ m$

where m is chosen such that for any position, p,

$-\ m < eval\ p < m$.

The list *alphas* is the list of alpha values that result after the search of each child. The recursion allows each element of this list to be used as a bound in the computation of the next element.

Parallel algorithms for minimax searching is an active area of research. This simple parallel alpha-beta algorithm is probably not the best solution to the problem. Our notation makes it easier to understand and verify algorithms, but fundamental problems of finding the best algorithm for a given problem remain.

5 PRIORITIES

If all computation is mandatory then scheduling is not a difficult problem, assuming we have a shared memory of sufficient size so we do not need to worry about communication between processors or running out of memory. There are simple scheduling algorithms [5] that are within a factor of two of being optimal, in the worst case, with respect to the time required to finish all computation.

This is not true with speculative computation. If we have n processors and a potential for parallelism that is much higher than n, then it is possible for a program to have a speedup approaching n if all processors do mandatory work or speculative work that will later become mandatory almost all of the time. On the other hand, if one processor does mandatory work and all other processors spend almost all of their time on speculative computation that will prove to be unneeded, then the speedup may not be much greater than one. Clearly, given a limited number of processors, we prefer to do that speculative work that is most likely to be required later. In general, it is not possible for the implementation to determine which speculative computations are the most worthwhile.

The solution to this problem is to let the programmer specify priorities. We can introduce a new function, *priority* of type *num* -> * -> *. The semantics of *priority* are

priority n $x = \bot$, **if** $n = \bot$
$= x$, **otherwise**

If priority is called within a speculative computation, it initiates a new speculative computation with priority n, where n is any number, to compute x. The higher the value of n, the higher the priority of the speculative computation. All speculative computations started with *spec* have higher priority than any started by *priority*. The priority of a speculative computation can not be changed, except that the speculative computation may become mandatory. In particular, when one speculative computation finds it needs the result of another, possi-

bly lower priority, speculative computation in order to proceed, it must wait for the second speculative computation to finish (unless, of course, the waiting computation becomes upgraded to mandatory.) Of course, if *priority* is invoked by a mandatory computation, then the resulting computation will immediately become mandatory, since it would not have been invoked if its result were not needed.

As an example, we can modify the least-cost search algorithm in Fig. 6 by changing the subexpression

(*foldr1 minimum.map search'.children*) *root*

to

(*priority* (−(*lower_bound root*)).
foldr1 minimum.map search'.children) *root*

so that nodes that are more promising (have a smaller lower bound) are expanded with higher priority. (Of course, we would want to factor out the computation of *lower_bound root* and compute it only once.)

6 FORMAL PROPERTIES

We will assume that the values used in improving values are finite in size and come from a flat domain. That is, values are either completely defined or are $\perp$. An example of a nonflat domain would be the domain of lazy lists. For example, the lazy list 1:$\perp$ has 1 as its first element, but any attempt to evaluate the tail of the list will result in a nonterminating computation or an undefined result.

If we allow values from a nonflat domain to be improving values, we have some awkward special cases. For example, with lists ordered in the obvious way,

break (*minimum* (*make* (1:$\perp$)) (*spec_max* (*make* (1:$\perp$)) (*make* (2:$\perp$)))) = $\perp$,

since the comparison of (1:$\perp$) with (1:$\perp$) will not terminate. On the other hand,

$$mini\ (1{:}\bot)\ (maxi\ (1{:}\bot)\ (2{:}\bot)) = 2{:}\bot$$

In this case, *maxi* is applied first and the nonterminating comparison does not arise. Similar problems can arise with infinite lists. We would like the improving value operations to be at least as well defined as the corresponding operations on ordinary values.

Sometimes it will be useful to have improving values where values come from a flat subdomain of a nonflat domain. For example, we might want to use improving lists in a context where we know that all lists will be finite and well defined. In section 4, we used improving ordered pairs. The domain of ordered pairs is not flat, since it includes elements such as $(1, \bot)$. However, we only used fully defined ordered pairs. In cases such as these, we must restrict ourselves to a flat subdomain. If type *improving* is implemented as suggested at the end of section 3, the restriction to finite, fully defined values can be enforced by requiring *make* to fully evaluate its argument. However, if the base domain is not a flat domain, the theoretical results that follow do not apply. For example, it will no longer be the case that $(break.make) = id$ if *make* forces complete evaluation of its argument. The results in the remainder of this section depend on values being finite and either fully defined or completely undefined before *make* is applied.

With the implementation of type *improving* given in Fig. 3, a number of different lists may all represent the same abstract *improving* value. Recall that lists are strictly increasing sequences of lower bounds. If a and b are any two well defined values with $a < b$ then it is not possible to distinguish $a{:}b{:}xs$ from $b{:}xs$ provide both are valid representations for *improving* values. Since *spec_max* and *minimum* both examine list elements sequentially, lists of the form $xs +\!\!+ \bot +\!\!+ ys$, where "++" is an infix append operator, cannot be generated, although lists of the form $xs +\!\!+ \bot$ can be produced. Finally, while both $[\bot]$ and $\bot$ are possible representations for *improving* values, $a{:}[\bot]$ is not a possible representation. For example, *spec_max* (*make* a) (*make* $\bot$) will produce

an *improving* value represented by $a{:}\bot$, even though *make* $\bot$ produces $[\bot]$. We cannot distinguish between the *improving* values represented by $\bot$ and $[\bot]$.

With these observations, we can divide representations into equivalence classes. We have five cases to consider: Fully defined finite lists, partial lists with at least one element, $\bot$, infinite lists with an upper bound, and infinite lists without an upper bound.

We will let $a!$ represent the class of all finite, fully defined lists with final value a. If xs is a member of the equivalence class $a!$, then we will represent the equivalence class that includes $xs +\!\!+ \bot$ by $a{+}?$. We will read $a!$ as "exactly a", and $a{+}?$ as "at least a". Both $[\bot]$ and $\bot$ belong to the same equivalence class which we will represent by $\bot$.

Finally, we have two cases to consider with infinite lists. An infinite list having no upper bound on the size of its elements is in an equivalence class represented by ∞. For example *foldr1 spec_max* (*map make* [1..]) will generate ∞. An infinite list where the elements have a least upper bound, a, is a member of the equivalence class $a{+}?$ considered above. Since the list is infinite and strictly increasing, the upper bound can never be reached.

Axioms for *improving* values are given in Fig. 9. In this figure, a and b may represent any values and x may represent any *improving* value. These axioms are complete, in that they fully specify the result for each possible combination of arguments.

We can prove that these axioms are satisfied by the implementation given in Fig. 3 using inductive proofs similar to those found in [**?**, Chapter 7]. For example, to prove

$$\mathit{minimum}\ a!\ b{+}? = \mathbf{if}\ a < b\ \mathbf{then}\ a!\ \mathbf{else}\ b{+}?$$

we need to consider a number of cases. The representation of $a!$ must be of the form $x{:}xs$ where xs may or may not be the empty list. Similarily, the representation of $b{+}?$ either must be of the form $y{:}ys{+}{+}\bot$, where ys may or may not be empty, or must be an infinite list bounded above by b. In the

$make\ \bot = \bot$

$make\ a = a!$

$break\ \bot = \bot$

$break\ a! = a$

$break\ a{+}? = \bot$

$break\ \infty = \bot$

$minimum\ x\ y = minimum\ y\ x$

$minimum\ \bot\ x = \bot$

$minimum\ \infty\ x = x$

$minimum\ a!\ b! = \textbf{if}\ a < b\ \textbf{then}\ a!\ \textbf{else}\ b!$

$minimum\ a!\ b{+}? = \textbf{if}\ a < b\ \textbf{then}\ a!\ \textbf{else}\ b{+}?$

$minimum\ a{+}?\ b{+}? = \textbf{if}\ a < b\ \textbf{then}\ a{+}?\ \textbf{else}\ b{+}?$

$spec\_max\ \bot\ x = \bot$

$spec\_max\ \infty\ x = \infty$

$spec\_max\ a{+}?\ x = a{+}?$

$spec\_max\ a!\ \bot = a{+}?$

$spec\_max\ a!\ \infty = \infty$

$spec\_max\ a!\ b! = \textbf{if}\ a < b\ \textbf{then}\ b!\ \textbf{else}\ a!$

$spec\_max\ a!\ b{+}? = \textbf{if}\ a < b\ \textbf{then}\ b{+}?\ \textbf{else}\ a{+}?$

Figure 9: Axioms for type *improving*.

case where $b+?$ is represented by an infinite list we must recognize that its representation is the limit of a sequence of representations for $c_1+?$, $c_2+?$, ..., where $c_i < c_{i+1} < a$ for all $i \geq 1$, but for any $d < b$, there exists $I \geq 1$ such that $i \geq I$ implies $c_i > d$. Finally, for each of these cases we must consider the cases $x < y$, $x = y$, and $x > y$.

With these axioms, we can prove a number of interesting properties. These include:

$$make\ (mini\ a\ b) = minimum\ (make\ a)\ (make\ b) \tag{1}$$

$$make\ (maxi\ a\ b) \sqsubset spec\_max\ (make\ a)\ (make\ b) \tag{2}$$

$$mini\ (break\ a)\ (break\ b) \sqsubset break\ (minimum\ a\ b) \tag{3}$$

$$maxi\ (break\ a)\ (break\ b) = break\ (spec\_max\ a\ b) \tag{4}$$

$$break.make = id \tag{5}$$

$$make.break \sqsubset id \tag{6}$$

We can use these properties to prove the correctness of efficient combinatorial search algorithms using *improving values*. To show that an implementation meets its specification, we show that *specification* $\sqsubset$ *implementation*. That is, where the specification is defined, the implementation must agree with it, but the implementation may be stronger. For example, we may take an exhaustive search algorithm as a specification for a more efficient search algorithm that avoids searching unnecessary subspaces. In these cases, the implementation must exceed the specification, because nonterminating computations in the pruned portion of the search space will be unencountered by the implementation, but would cause the specification to fail to terminate.

If we take the algorithm in Fig. 4 as a specification for the more efficient least-cost search algorithm of Fig. 6, then we can easily prove the least-cost

search algorithm correct. First we recall that the algorithm in Fig. 5 is equivalent to the one in Fig. 4. Using properties 6, 1, 2 and 5 above, it is easy to show that the least-cost search algorithm meets its specification.

7 CONCLUSION

We have seen that the polymorphic abstract data type *improving* allows us to express various combinatorial algorithms in a manner that is simpler than most previous expressions, yet at the same time introduces parallelism into the problem. Furthermore, the type *improving* has an axiomatic specification from which we can derive several important properties, which in turn can be use to prove the correctness of programs using the type.

As an example, we presented a short program for a parallel least- cost search that is optimal on a single processor and can make good use of any number of processors. A correctness proof of the function was outlined and can be easily completed by the reader.

Acknowledgement Discussions with N. S. Sridharan on methods for maintaining bounds in parallel combinatorial search algorithms contributed to the development of the idea of improving values. The author would also like to thank Simon Peyton Jones and Ken Jackson for numerous helpful comments on earlier drafts of this paper. In particular, Ken Jackson made an observation that simplified the set of axioms considered in section 6.

References

[1] F. Warren Burton. Controlling speculative computation in a parallel functional programming language. In *Proceedings of The Fifth International Conference on Distributed Computing Systems*, pages 453–458, Denver, Colorado, May 1985.

[2] F. Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Trans. Comput.*, C-34(12):1190–1193, Dec. 1985.

[3] F. Warren Burton. Functional programming for concurrent and distributed computing. *Comput. J.*, 30(5):437–450, Oct. 1987.

[4] F. Warren Burton. Encapsulating nondeterminacy in an abstract data type. *Journal of Functional Programming*, to appear in about January 1991.

[5] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, March 1989.

[6] D. H. Grit and R. L. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Trans. Prog. Lang. and Syst.*, 3(1):49–59, Jan. 1981.

[7] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[8] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proc. 1982 ACM Symposium on LISP and Functional Programming*, pages 168–178, Pittsburgh, Penn., Aug. 1982.

[9] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[10] David A. Turner. Functional programs as executable specifications. In C. A. R. Hoare J. Shepherdson, editor, *Mathematical logic and programming languages*, pages 29–54. Prentice-Hall, 1985.

[11] David A. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, Dec. 1986.